UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

ESTUDIO DE DIVISORES BINARIOS EN HARDWARE

Autor: Cristian Roibu Roibu
Tutor: Federico Favaro
Ponente: Eduardo Boemo

Junio 2020

ESTUDIO DE DIVISORES BINARIOS EN HARDWARE

Autor: Cristian Roibu Roibu
Tutor: Federico Favaro
Ponente: Eduardo Boemo

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Abril 2020

Resumen

El objetivo de este trabajo fin de grado es estudiar algunas opciones para divisores binarios enteros en hardware.

El estudio forma parte de una investigación para desarrollar, en hardware, un dispositivo inalámbrico de bajo consumo para obtener señales electroencefalográficas (EEG). El proyecto se enmarca en una línea de investigación del Instituto de Ingeniería Eléctrica de la Universidad de la República de Uruguay sobre aplicaciones de bajo consumo de energía en FPGAs.

Este trabajo de fin de grado se centra en estudiar diferentes posibilidades para realizar de una forma eficiente la división entera en hardware, concretamente en una FPGA de **Intel-Altera** (Intel adquirió Altera en 2015). Para ello se procederá a analizar distintas implementaciones y caracterizar su rendimiento en términos de latencia, frecuencia máxima de operación, utilización de recursos y consumo de energía.

A partir de esto se aprenderá el manejo de herramientas **Altera** y se reforzará el conocimiento en el lenguaje de descripción de hardware VHDL.

Palabras clave

Aritmética, división, dividendo, divisor, cociente, resto, flanco de reloj, área, latencia, frecuencia, algoritmo, bit, FPGA, pipeline, resta, desplazamiento, restauración, nonperforming, redundancia, lista de sensibilidad.

Abstract

The objective of this final degree project is to study different implementations for integer binary division in hardware.

The study is part of an investigation to develop, in hardware, a wireless device of low energy to obtain electroencephalographic signals (EEG). The project is part of a research line of the Institute of Electrical Engineering of the University of the Republic of Uruguay on applications of low energy consumption in FPGAs.

This final degree project in studying different possibilities to efficiently perform the unsigned division in hardware, specifically in an **Intel-Altera** FPGA (Intel bought Altera in 2015). This will proceed to analyse different implementations and characterize their performance in terms of latency, maximum frequency of operation, use of resources and energy consumption.

From this, the management of **Altera** tools will be learned and knowledge in the VHDL programming language will be reinforced.

Keywords

Arithmetic, division, dividend, divider, quotient, remainder, clock edge, area, latency, frequency, algorithm, bit, FPGA, pipeline, subtraction, shift, restoration, nonperforming, redundancy, sensibility list.

Agradecimientos

Mis agradecimientos van para todas las personas que han contribuido con su tiempo, en mayor o menor medida, a ayudarme en la realización de este Trabajo Fin de Grado.

Por su puesto a mis padres, por ser mi apoyo diario, a mis amigos, por estar siempre ahí y sobre todo por la parte que les toca a mis compañeros de universidad durante todos estos años Jesús Romano, Jorge Rubio Zaballos, David Vázquez, Lucas Caporale y Javier Matellano.

Por último, agradecer también a mi ponente, Eduardo Boemo, por haber estado siempre disponible para ayudarme y haberme dedicado su tiempo en algo tan importante para mí como es mi TFG.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

INDICE DE GRAFICAS

1 Introducción

El objetivo de este Trabajo Fin de Grado (TFG) reside en presentar un pequeño apartado teórico sobre divisores binarios en hardware. A continuación, se exponen las motivaciones y los antecedentes que han impulsado dicho proyecto, así como una introducción de este.

1.1 Motivación

El estudio forma parte de un proyecto en el que se busca desarrollar, en hardware, un dispositivo inalámbrico de bajo consumo para la obtención de señales electroencefalográficas (EEG). El proyecto se enmarca en una línea de investigación del Instituto de Ingeniería Eléctrica de la Universidad de la República (UDELAR) de Uruguay sobre aplicaciones de bajo consumo de energía en FPGAs.

Se busca que el sistema sea capaz de adquirir hasta 64 canales, a 16 bits por canal y con frecuencias de muestreo superiores a 1000 muestras por segundo. Esto implica una mínima tasa de datos a la salida del transmisor inalámbrico de:

Throughput = 64 canales x 16 bits x 1000 muestras/s = 1.02 Mbps

Esta tasa impone un elevado consumo de energía en sistemas que se alimentan mediante batería. Por ello, el sistema tiene la capacidad de comprimir las señales adquiridas. El algoritmo de compresión elegido es de baja complejidad, utiliza aritmética de enteros y logra excelentes tasas de compresión para señales EEG.

Anteriormente ya se realizó una primera implementación del algoritmo de compresión en hardware usando VHDL y actualmente nos encontramos en la etapa de optimización del diseño para disminuir el consumo de energía y la utilización de recursos (área).

La implementación del algoritmo en hardware presenta la ventaja, respecto a la versión para el microcontrolador, de que permite que todos los canales puedan ser procesados en paralelo. Esto aumenta la velocidad de procesamiento, pero por otro lado también aumenta la cantidad de recursos de área necesarios para el diseño.

El algoritmo de división binaria utiliza fundamentalmente operaciones aritméticas fáciles de implementar en hardware, como sumas y desplazamientos, pero necesita una división por cada muestra. Esto quiere decir que una implementación en la que se comprimen en paralelo todos los canales necesitaría tantos bloques de divisores como canales haya. Dado que el bloque divisor en hardware utiliza gran cantidad de recursos, su optimización presenta bastante margen de mejora.

1.2 Objetivos

El objetivo en el que se enfoca este TFG se basa en explorar diferentes posibilidades para realizar de una forma eficiente la división entera en hardware, concretamente en una FPGA. Para ello se procederá a analizar distintas implementaciones y caracterizar su rendimiento en términos de latencia, frecuencia máxima de operación, utilización de recursos y consumo de energía. Se concluirá en base a los datos obtenidos cual es la mejor alternativa para nuestro proyecto.

1.3 Organización de la memoria

La memoria se constituye por capítulos, y éstos contienen diferentes secciones:

- En el capítulo 2 se introducen los conceptos básicos de un bloque divisor, así como aquellas características que sería interesante implementar en nuestro sistema. Más adelante, se especifican las diferentes alternativas teóricas que presenta la operación aritmética de la división binaria.

- En el capítulo 3 se traducen los distintos métodos analizados teóricamente a lenguaje VHDL para su posterior análisis.

- En el capítulo 4 se analizan los resultados obtenidos en las distintas simulaciones

- Por último, en el capítulo 5 se detallan las conclusiones obtenidas de este TFG.

2 Estado del arte

Se desea realizar un bloque que implemente la división binaria entre enteros, es decir, que reciba entradas D y d y tenga como salidas q y r tal que $\boldsymbol{D = d \times q + r}$, con $q \leq D$ y $r < d$. El diagrama del bloque que se quiere implementar es el siguiente:

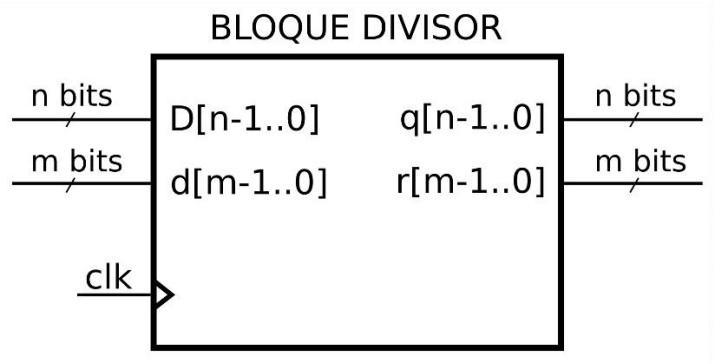


Figura 1: Bloque Divisor

A continuación, se presentan las principales características que sería interesante tener en cuenta en el bloque divisor y algunas consideraciones necesarias para facilitar su implementación:

1. La operación por realizar es una división entera. Esto quiere decir que las salidas (tanto el cociente como el resto) serán números enteros.
2. El diseño debería ser parametrizable en cantidad de bits de entradas y salidas (n y m en el diagrama).
3. El signo del divisor será siempre positivo, lo que simplifica en cierta medida el diseño. El signo del dividendo puede ser positivo o negativo.
4. La aplicación solo utiliza el cociente de la división, por lo que no será necesario obtener el resto a la salida. Para ciertos algoritmos de división binaria, esto implica realizar un número menor de operaciones, lo que a su vez repercute directamente en el tamaño y la velocidad del circuito.

La aplicación donde se pretende utilizar el divisor está compuesta de varias etapas. Por lo general, se requiere que cada etapa finalice para que pueda comenzar la siguiente. Por ello, es importante que cada bloque emplee un tiempo de procesamiento lo más corto posible. Esto es lo mismo que decir que la cantidad de ciclos de reloj que son necesarios para obtener un dato nuevo sea la menor posible. Por otro lado, nos encontramos con que es recomendable que la frecuencia máxima de operación sea relativamente alta. Nuevamente es importante destacar que otro de los principales requerimientos es la búsqueda de un diseño que sea óptimo en utilización de recursos (área).

2.1 Algoritmos de división

2.1.1 Generalidades

La operación de división es una de las 4 operaciones aritméticas básicas y ocurre con mucha menor frecuencia que las demás. Normalmente, se realiza en la mayoría de los computadores mediante un circuito sumador/restador y algún algoritmo asociado.

Dados dos operandos (el dividendo D y el divisor d) el objetivo de esta operación es calcular el cociente (q) y el resto (r) para que:

$$D = d \times q + r \qquad con\ 0 < r \leq d$$

En este estudio de algoritmos de división se han incluido 5 circuitos distintos. Aunque no todos se implementarán sí que es interesante comentar todos para tener una foto fija de cuáles son las distintas alternativas existentes. La diferencia fundamental entre ellos radica en la forma de manejar los restos parciales que se van obteniendo en el proceso de división.

Por último y aunque para el caso de estudio que nos ocupa no es necesario, cabría destacar que cuando nos encontramos con un dividendo y un divisor con signo distinto, el cociente final será el complemento a 2 del cociente hallado, otorgándole al mismo un bit de signo igual a 1. En caso contrario, cuando dividendo y divisor tienen ambos el mismo signo, el cociente pasa a dejarse tal cual se ha obtenido añadiendo un bit de signo igual a 0.

En la figura 2, se muestra un esquema básico de lo que sería un hardware de división binaria:

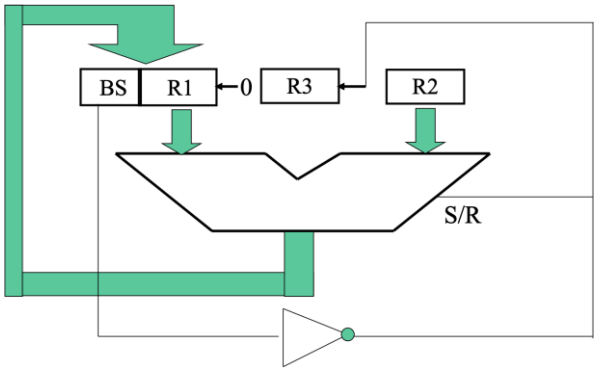


Figura 2: Esquema básico de división binaria en hardware

2.1.2 Método de lápiz y papel

En este método se sigue el siguiente algoritmo:

1. Tomamos inicialmente, partiendo desde los bits más significativos, tantos bits del número total de bits del dividendo como tenga el divisor.
2. Si la parte del dividendo es menor que el divisor, ponemos un 0 en el cociente y bajamos otro bit del dividendo.
   Por el contrario, si la parte del dividendo es mayor, ponemos un 1 en el cociente y realizamos la resta.

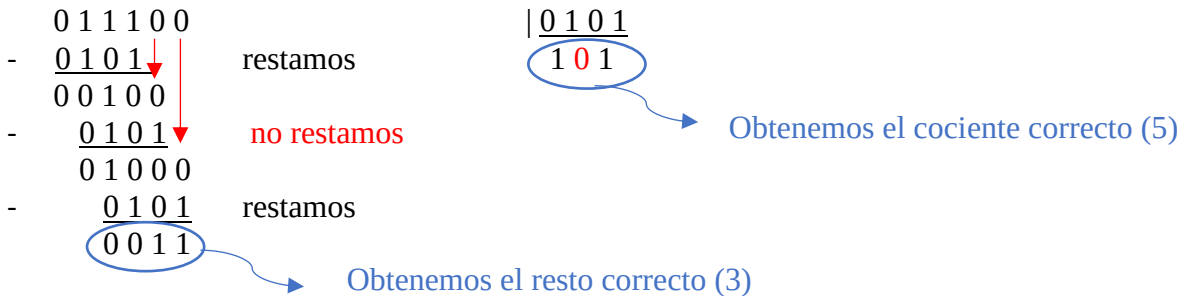
3. Repetimos el proceso de manera que vayamos obteniendo los resultados parciales de cada resta.

4. En cada nueva iteración, nos vamos desplazando hacia la derecha y vamos bajando de uno en uno los bits que quedan en el dividendo.

Podemos ver un ejemplo de esto en la siguiente división:

D = 28 = 011100
d = 5 = 0101

En este caso el cociente tendría que ser 5 y el resto 3. Comprobamos que estamos en lo cierto:



Como podemos observar, este método es el que más se asemeja al método tradicional de divisiones con números naturales. Es por ello por lo que, si nos centramos en la implementación de un circuito digital divisor, este método se aleja de la versión óptima.

El principal problema que plantea este método a la hora de extrapolarlo al mundo de las operaciones digitales radica en el hecho de que nos encontramos con una alineación de los bits que no puede ser hecha de manera automatizada. Necesitamos darnos cuenta de cuáles son los bits más significativos para, a partir de ahí, comenzar a hacer las restas parciales.

Es por ello por lo que definitivamente este método queda descartado para el objetivo principal de este trabajo de fin de grado. A continuación, vamos a pasar a analizar otros métodos alternativos que podrían dar solución a los problemas que este primer método analizado plantea.

2.1.3 División con restauración

Para este método es necesario realizar comparaciones constantes entre dividendo y divisor. Estas comparaciones se basan fundamentalmente en comprobar si el resultado de cada resta parcial es positivo o negativo.

Si el resultado de la resta parcial es positivo, se introduce un 1 en el cociente, se coge (se tiene en cuenta, igual que en la división manual) el bit siguiente del dividendo y se desplaza el divisor hacia la derecha para volver a hacer la siguiente resta.
Pero, por el contrario, si el resultado parcial es negativo, introducimos un 0 y, partiendo del dividendo del que se ha realizado la resta y no del resultado parcial, cogemos un nuevo bit y desplazamos hacia la derecha el divisor.

Al igual que en el método tradicional, vamos repitiendo estos desplazamientos después de cada resta hasta llegar al final de la cadena de bits del dividendo (siempre de izquierda a derecha).

En la figura 3, se muestra un pequeño diagrama que vendría a ilustrar lo anteriormente citado.

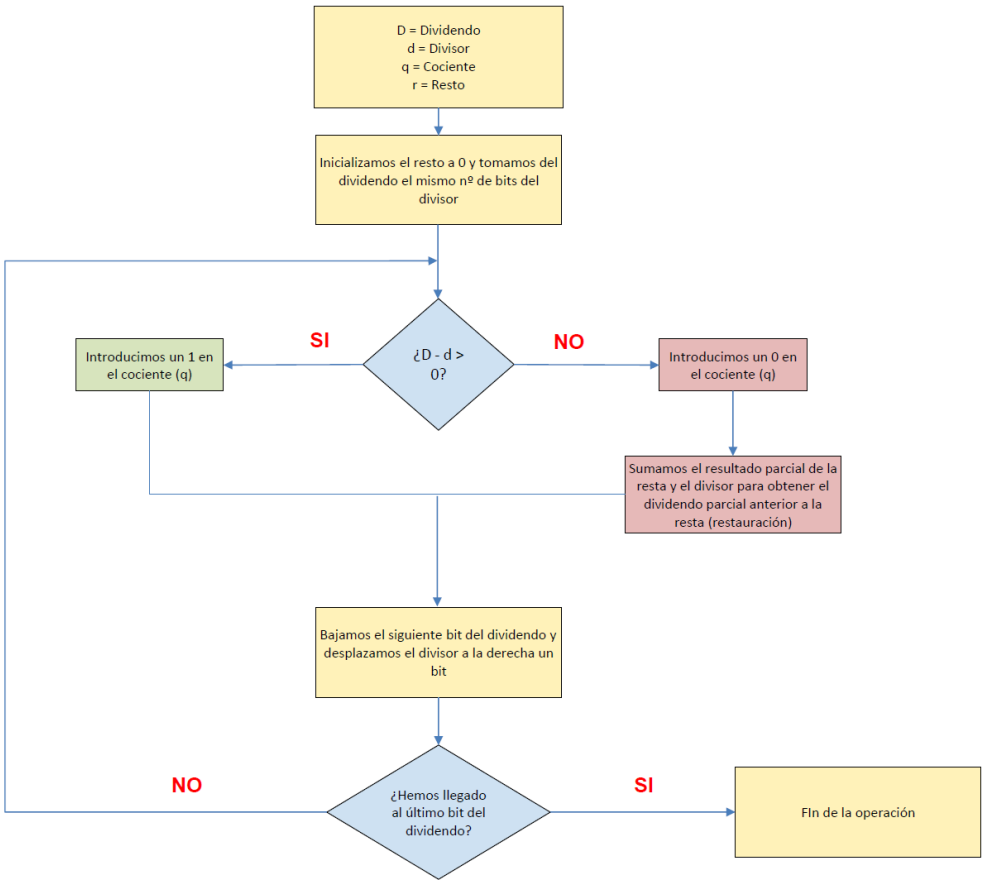
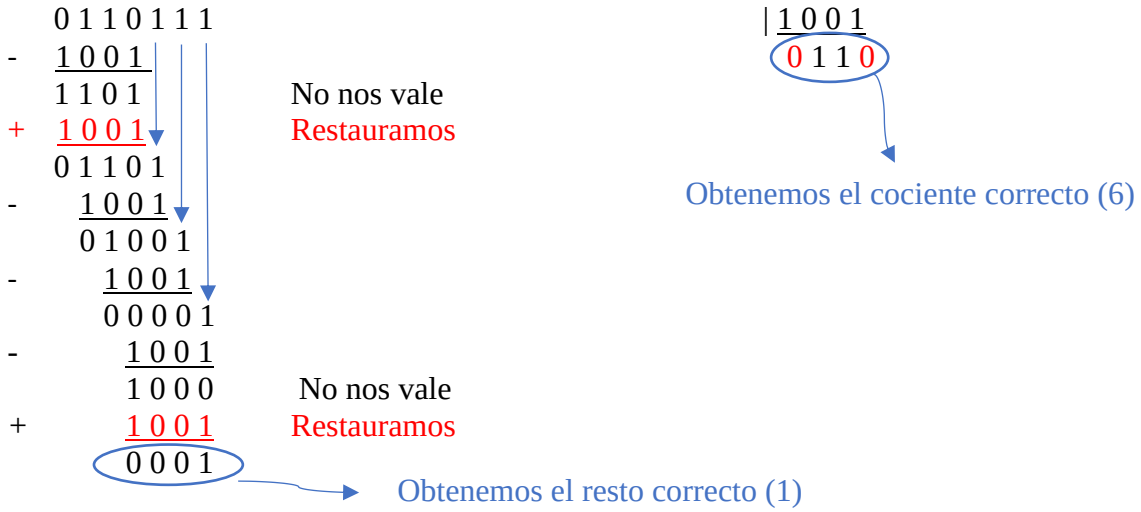


Figura 3: Esquema de la división con restauración

A continuación, pasamos a realizar la siguiente operación a modo de ejemplo:

D = 55 = 0110111
d = 9 = 1001



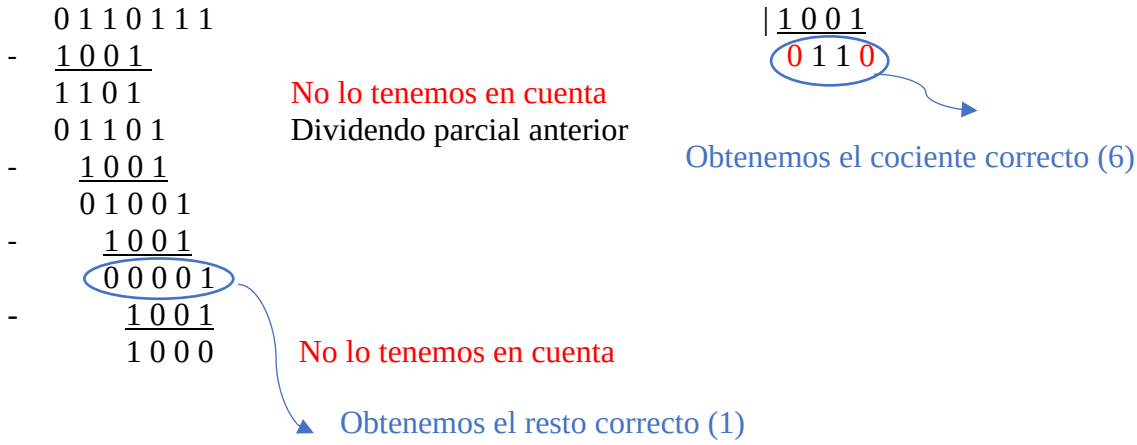
2.1.4 Algoritmo nonperforming

Este método se podría considerar como una variación del método con restauración anteriormente explicado.

A través del algoritmo de nonperforming conseguimos mejorar sustancialmente la velocidad de ejecución del algoritmo ya que eliminamos la suma que se realiza posteriormente a la detección de un resultado parcial negativo. En su lugar lo que se hace es recuperar el dividendo parcial anterior a la resta y a partir de él, bajar el siguiente bit del dividendo y desplazar el divisor a la derecha un bit.

A continuación, pasamos a realizar la siguiente operación a modo de ejemplo:

D = 55 = 0110111
d = 9 = 1001



2.1.5 División sin restauración

Para este otro método, lo que se pretende es agilizar todo el algoritmo, eliminando la restauración que se realiza cuando obtenemos un resultado negativo en la comparación por resta.

En la figura 4, se muestra el diagrama de flujo de cómo sería el proceso en este caso:

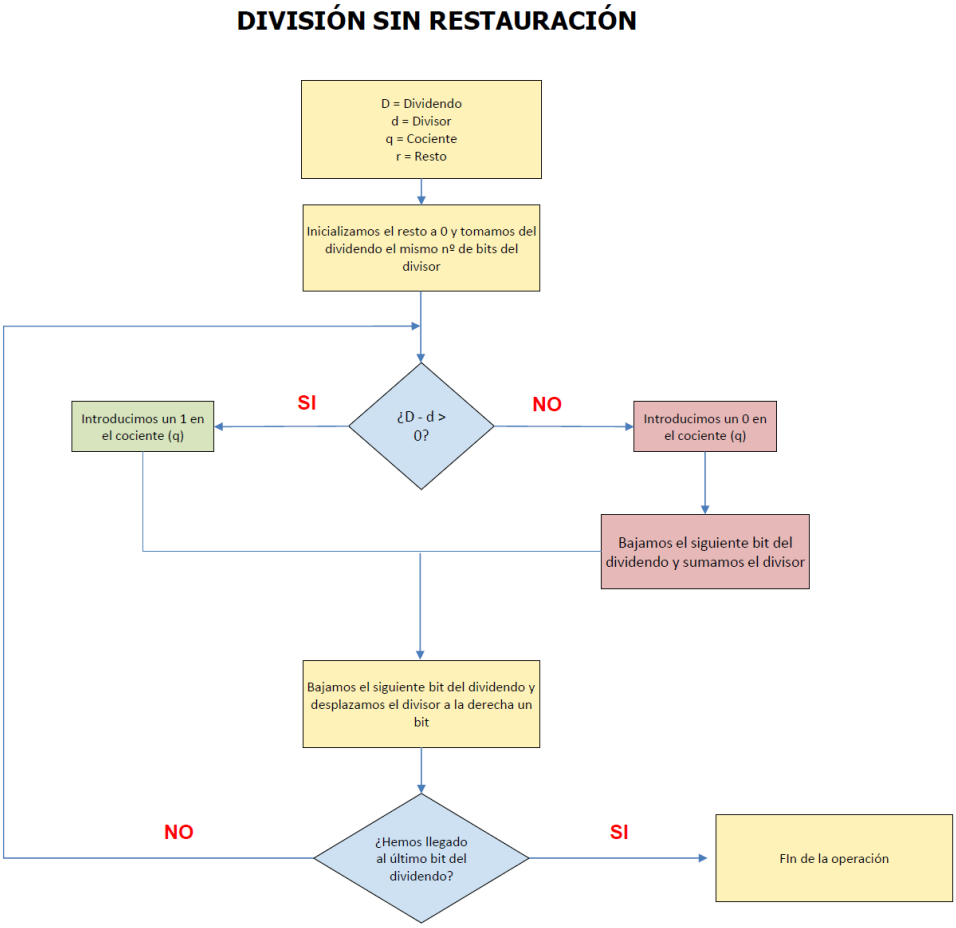


Figura 4: Esquema de la división sin restauración

Con esto lo que pretendemos es obtener en una sola operación los restos parciales que vayan apareciendo durante el proceso de la división.

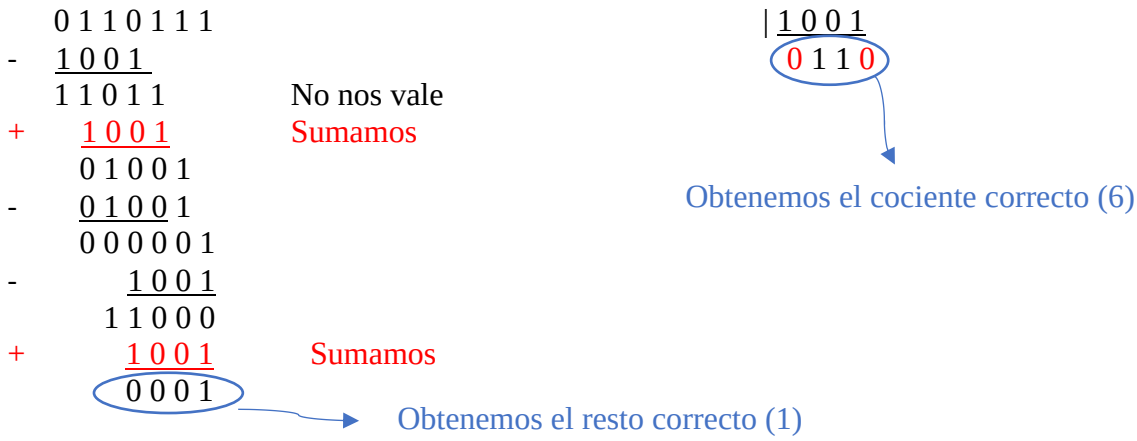
Para ello debemos tener en cuenta lo siguiente:

- Si el resultado parcial de la resta es positivo, introducimos un 1 en el cociente (por la derecha), bajamos el siguiente bit del dividendo y desplazamos el divisor hacia la derecha un bit para realizar la siguiente resta.

- Si, por el contrario, el resultado parcial de la resta es negativo, sumamos ese resultado al divisor, introduciendo un 0 en el cociente (también por la derecha), bajamos un bit del dividendo y desplazamos el divisor hacia la derecha un bit, para, en este caso, realizar la siguiente resta correspondiente al proceso a partir de la suma obtenida y el bit del dividendo bajado.

De esta forma quedaría la división sin restauración tomando el mismo ejemplo que en el caso con restauración.

D = 55 = 0110111
d = 9 = 1001



2.1.6 Uso de redundancia

Para explicar el uso de redundancia debemos partir de una división sin restauración en la que en un momento cualquiera de la operación obtengamos un resto parcial igual a 0. Pues bien, en este caso, al volver a realizar la siguiente resta, obtendríamos un nuevo resto parcial igual a -d (es decir, el negativo del divisor). En este caso, lo siguiente que tocaría hacer sería desplazar y realizar una suma (ya que, como hemos visto anteriormente en el algoritmo de división sin restauración cuando se introduce un 0 en el cociente la siguiente operación a realizar es una suma y no una resta). No obstante, esto no tiene sentido ya que volveríamos a obtener un nuevo resto parcial igual a -d.

Es por ello por lo que el uso de redundancia nos dice que, en estos casos, podemos desplazar directamente y omitir la operación de suma estipulada para estas situaciones en la división sin restauración.

Veámoslo nuevamente con un ejemplo:

Supongamos que tenemos la siguiente división:

D = 205 = 011001101
D = 6= 0110

```
    0 1 1 0 1 1 1                                  | 0 1 1 0
-   0 1 1 0
    0 0 0 0
-   0 1 1 0
    1 0 1 0                  -D
  1 0 1 0 0                  Doblamos (desplazamos) = -2D = -12
+   0 1 1 0
    1 0 1 0                  -D
      .
      .
      .
```

2.1. 7 Algoritmo de división rápida SRT

Como hemos visto en los métodos anteriores, fundamentalmente para el algoritmo de división sin restauración, la idea es agilizar el proceso de división obteniendo así de una forma más rápida cociente y resto. No obstante, para esto, hemos visto que es necesario realizar una suma o resta en cada iteración.

Lo que se pretende conseguir con el algoritmo SRT es una mayor velocidad de operación que en el algoritmo de división sin restauración, reduciendo con ello el número de sumas y restas realizadas.

Para ello, lo primero que hay que tener en cuenta es que este método será válido siempre que el dividendo sea menor que el divisor cumpliendo con estos dos rangos:

$$-2^{n-1} \leq D < 2^{n-1}$$
$$2^{n-1} \leq d < 2^{n}$$

Esto es importante ya que de otra forma el algoritmo no se comporta de manera óptima y los resultados obtenidos no se corresponden con los esperados.

Por otro lado, es necesario tener en cuenta el número de bits de precisión de la operación. Aunque en principio este número podrá ser el que nosotros queramos, cuanto mayor sea, más precisa será nuestra división y por ende más tardará en ejecutarse el algoritmo. Para las operaciones que se van a realizar en este proyecto, estableceremos una precisión en este algoritmo de ***p=8 bits***.

El algoritmo empieza multiplicando por 2 el dividendo y comparándolo con el extremo superior del rango establecido inicialmente. Para el valor obtenido tras multiplicar por 2 el dividendo, aparecen 3 posibles valores para el cociente (-1,0 y 1) que pueden ser usados.

Asimismo, dependiendo del valor del cociente la operación posterior a realizar para obtener el resto será diferente. La regla de decisión del cociente y resto es la siguiente:

- Si **2*resto_parcial(i-1) < - 2^n-1**, q(i) = -1
    - Resto(i) = 2* resto_parcial (i-1) + d
- Si **-2^n-1≤ 2*restoparcial(i-1) < 2^n-1**, q(i) = 0
    - Resto(i) = 2* resto_parcial (i-1)
- Si **2*resto_parcial(i-1) ≥ 2^n-1**, q(i) = 1
    - Resto(i) = 2* resto_parcial (i-1) - d

Ahora bien, como debido a esta regla algunos bits del cociente aparecerán como números con signo, una vez termine el proceso habrá que realizar una conversión al formato sin signo.

La conversión final del cociente será la siguiente:

$$Q\_final = Q\_pos - Q\_neg$$

Por último, si quisiéramos comprobar que efectivamente el resultado de nuestra operación es el correcto, deberíamos utilizar la siguiente fórmula:

$$2^{p} \times D = q \times d + r$$

A continuación, aparece un ejemplo con el que podemos demostrar esto:

Supongamos que queremos dividir **5 entre 10 con una precisión de p=4**. Lo primero de todo sería comprobar que efectivamente, puesto que ambos números se pueden representar con 4 bits, el criterio de rangos se cumple.

$$-2^{3} \leq 5 < 2^{3} = -8 \leq 5 < 8$$

$$2^{3} \leq 10 < 2^{4} = 8 \leq 10 < 16$$

Una vez hemos comprobado que la operación se podría realizar sin problema teniendo en cuenta el algoritmo, comenzamos a operar:

Iteración	Comparación	Cociente	Operación	Resto
Inicialización	r(0) = 5			
0	2*r(0) = 10 < 16	q(1) = 0	r(1) = 2*r(0)	10
1	2*r(1) = 20 > 16	q(2) = 1	r(2) = 2*r(1) - d	20 – 10 = 10
2	2*r(2) = 20 > 16	q(3) = 1	r(3) = 2*r(2) - d	20 – 10 = 10
3	2*r(3) = 20 > 16	q(4) = 1	r(4) = 2*r(3) - d	20 – 10 = 10

Tabla 1: Ejemplo División SRT

Por último, realizamos la comprobación para asegurarnos que la operación se ha realizado correctamente.

$$2^{4} * 5 = 7\ (0111) * 10 + 10$$

$$16 * 5 = 70 + 10$$

$$80 = 80$$

Por lo tanto, podemos concluir que la operación es correcta.

Cabe destacar que, si esta operación la quisiéramos realizar con una precisión de p=8, el resultado sería el siguiente:

Iteración	Comparación	Cociente	Operación	Resto
Inicialización	r(0) = 5			
0	2*r(0) = 10 < 16	q(1) = 0	r(1) = 2*r(0)	10
1	2*r(1) = 20 > 16	q(2) = 1	r(2) = 2*r(1) - d	20 – 10 = 10
2	2*r(2) = 20 > 16	q(3) = 1	r(3) = 2*r(2) - d	20 – 10 = 10
3	2*r(3) = 20 > 16	q(4) = 1	r(4) = 2*r(3) - d	20 – 10 = 10
4	2*r(4) = 20 > 16	q(5) = 1	r(5) = 2*r(4) - d	20 – 10 = 10
5	2*r(5) = 20 > 16	q(6) = 1	r(6) = 2*r(5) - d	20 – 10 = 10
6	2*r(6) = 20 > 16	q(7) = 1	r(7) = 2*r(6) - d	20 – 10 = 10
7	2*r(7) = 20 > 16	q(8) = 1	r(8) = 2*r(7) - d	20 – 10 = 10

Tabla 2: Ejemplo 2 División SRT

Si volvemos a realizar la comprobación final vemos que:

2^8 * 5 = 127 (01111111) * 10 + 10

256 * 5 = 1270 + 10

1280 = 1280

3 Análisis práctico de algoritmos

Una vez vistos los principales algoritmos de división binaria, en esta nueva sección vamos a pasar a analizar el comportamiento de algunos de ellos para determinar cuál es el más adecuado para implementar en nuestro hardware.

3.1 Elementos empleados

3.1.1 Herramienta de diseño y análisis

Para ello, la herramienta software utilizada ha sido **Quartus Prime 18.1 Lite Edition**. Esta es una herramienta de diseño de FGPAs que nos va a permitir implementar partiendo de VHDL, los diferentes algoritmos analizados para posteriormente obtener parámetros de rendimiento como la frecuencia máxima de operación, utilización de recursos y el área.

3.1.2 Elementos hardware empleados

Para todas las simulaciones, el modelo de placa que vamos a considerar en la herramienta es la **DE0-CV**. El chip o la FPGA es la **Cyclone V 5CEBA4F23C7N**, como se puede ver en la figura 5.

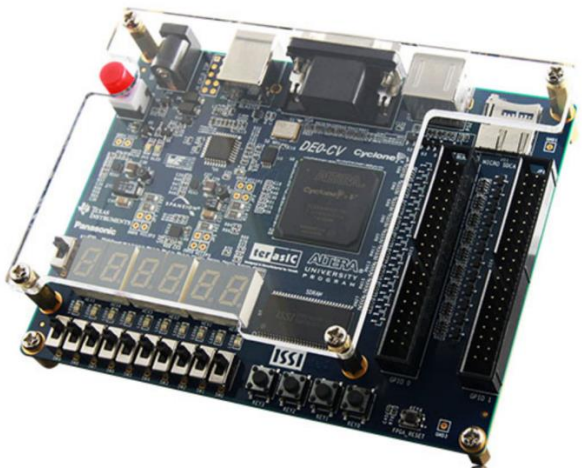


Figura 5: FPGA Cyclone V 5CEBA4F23C7N

En la tabla 3, se detallan las especificaciones más destacables de la placa empleada.

Series	**Cyclone® V E**
Part Status	Active
Number of LABs/CLBs	18480
Number of Logic Elements/Cells	49000
Total RAM Bits	3464192
Number of I/O	224
Voltage - Supply	1.07V ~ 1.13V
Mounting Type	Surface Mount
Operating Temperature	0°C ~ 85°C (TJ)
Package / Case	484-BGA
Supplier Device Package	484-FBGA (23x23)
Base Part Number	**5CEBA4**

Tabla 3: Especificaciones Cyclone V

3.2 Implementación de algoritmos

En esta sección vamos a analizar el comportamiento de algunas de las distintas alternativas de división binaria vistas en el apartado teórico. Para ello, vamos a comenzar codificando cada algoritmo en VHDL para posteriormente analizar su comportamiento en los términos que nos afectan para nuestro estudio.

Nota 1: En caso de querer consultarse, los códigos empleados para cada uno de los algoritmos a los que se hace referencia en este apartado se encuentran en la sección 'Anexos'.

3.2.1 División con restauración

En la figura 6 podemos ver el diseño obtenido a partir de la herramienta de Quartus RTL Viewer con las distintas celdas divisoras. Es importante mencionar aquí, que con respecto a otro diseños en los que no se tiene en cuenta la señal de reloj, en este caso una de las cosas que realmente modifica como se sintetiza el circuito es la instrucción "if rising_edge(clk) then.."

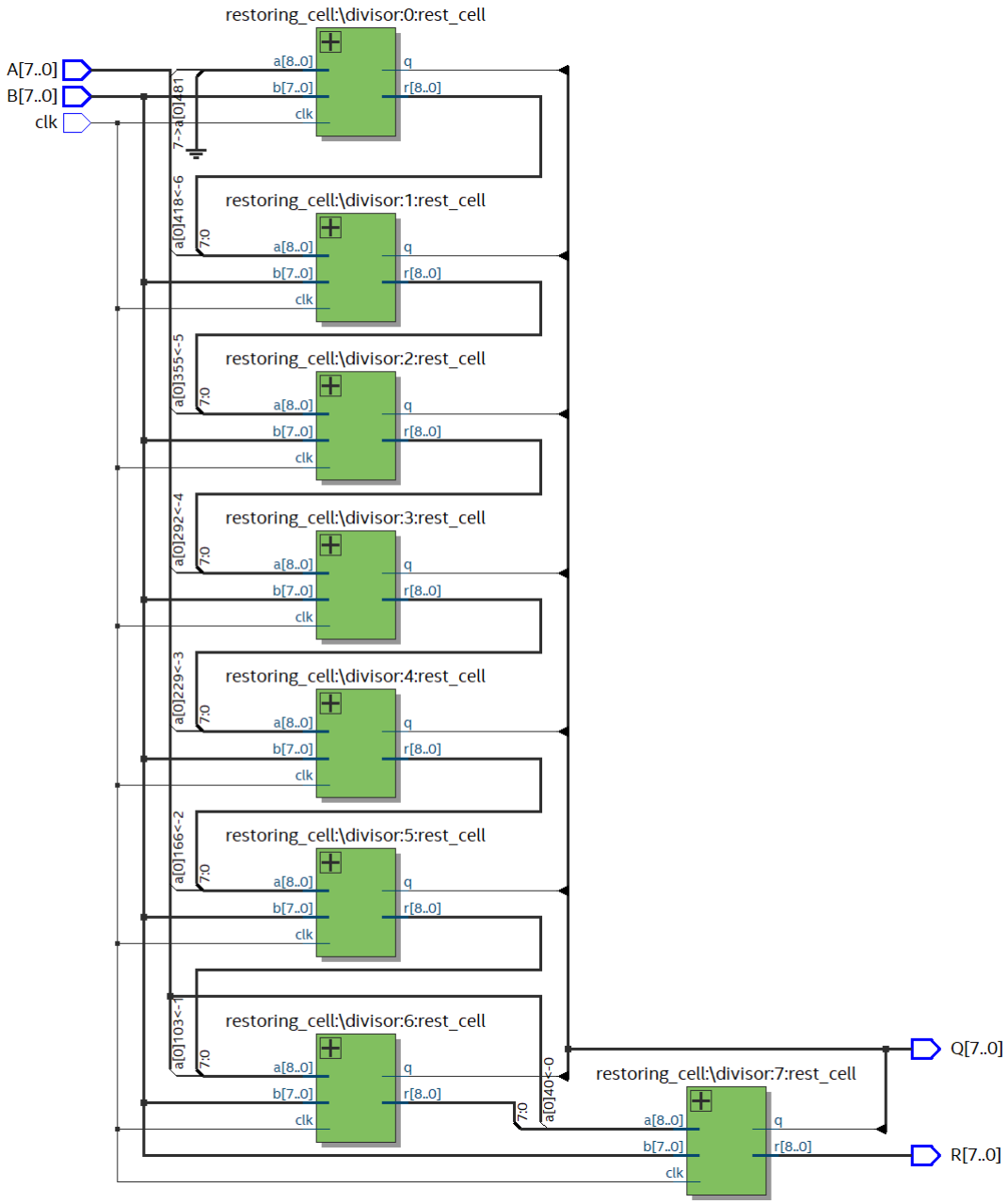


Figura 6: Diseño división con restauración con herramienta Quartus

En las figuras 7, 8 y 9, tenemos un resumen de los principales parámetros de utilización y recursos que consume este primer algoritmo analizado, así como la frecuencia máxima de trabajo.

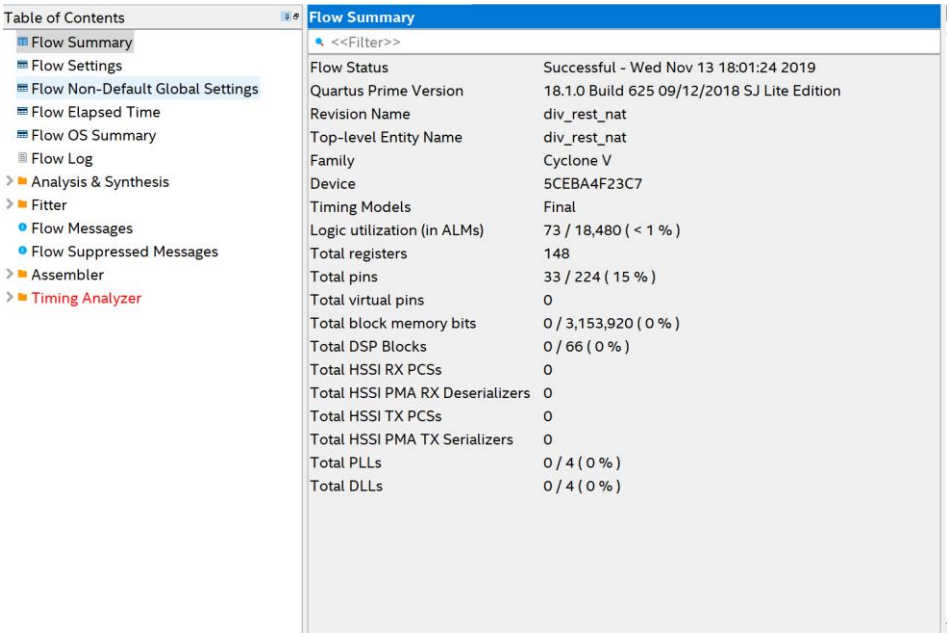


Figura 7: Utilización división con restauración

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	72
2		
3	⌄ Combinational ALUT usage for logic	88
1	-- 7 input functions	0
2	-- 6 input functions	0
3	-- 5 input functions	0
4	-- 4 input functions	0
5	-- <=3 input functions	88
4		
5	Dedicated logic registers	144
6		
7	I/O pins	33
8		
9	Total DSP Blocks	0
10		
11	Maximum fan-out node	clk~input
12	Maximum fan-out	144
13	Total fan-out	666
14	Average fan-out	2.23

Figura 8: Parámetros división con restauración

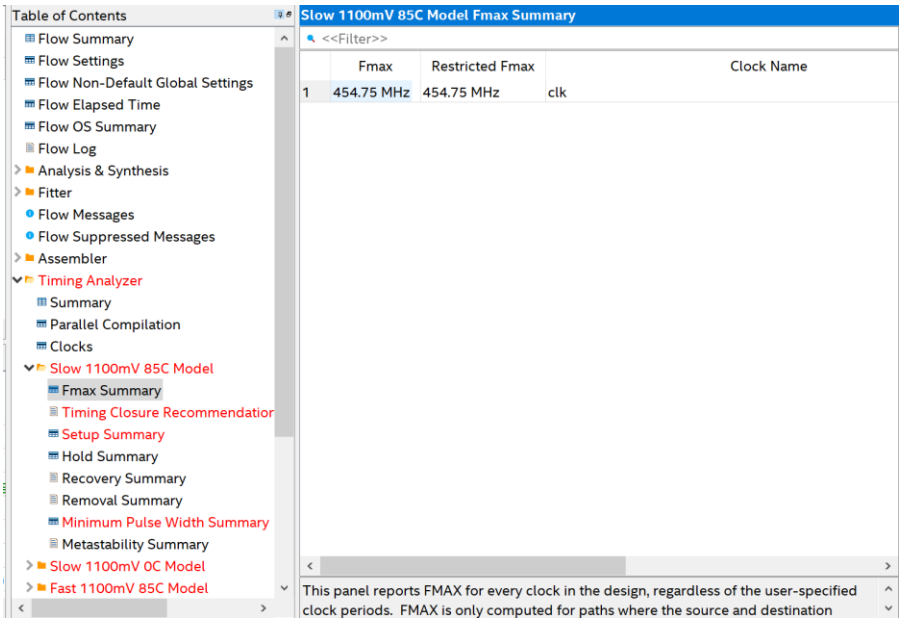


Figura 9: Frecuencia máxima división con restauración

3.2.2 División sin restauración

En la figura 10, podemos ver el diseño implementado a partir del código VHDL, en este caso para la división sin restauración.

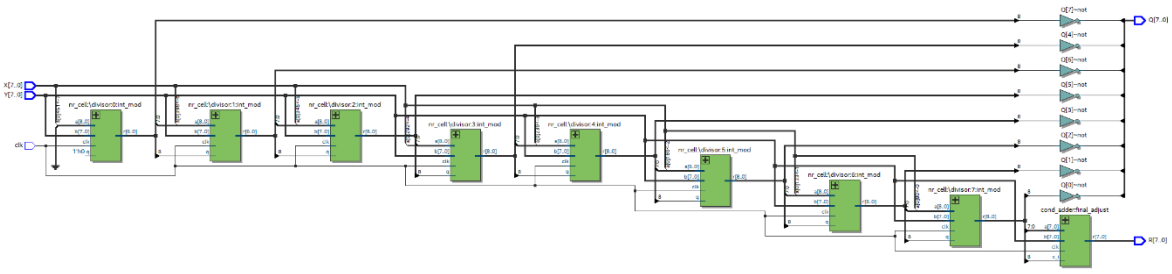


Figura 10: Diseño división sin restauración herramienta Quartus

En las figuras 11, 12 y 13 nuevamente aparecen los valores en términos de recursos y utilización para el algoritmo de división sin restauración.

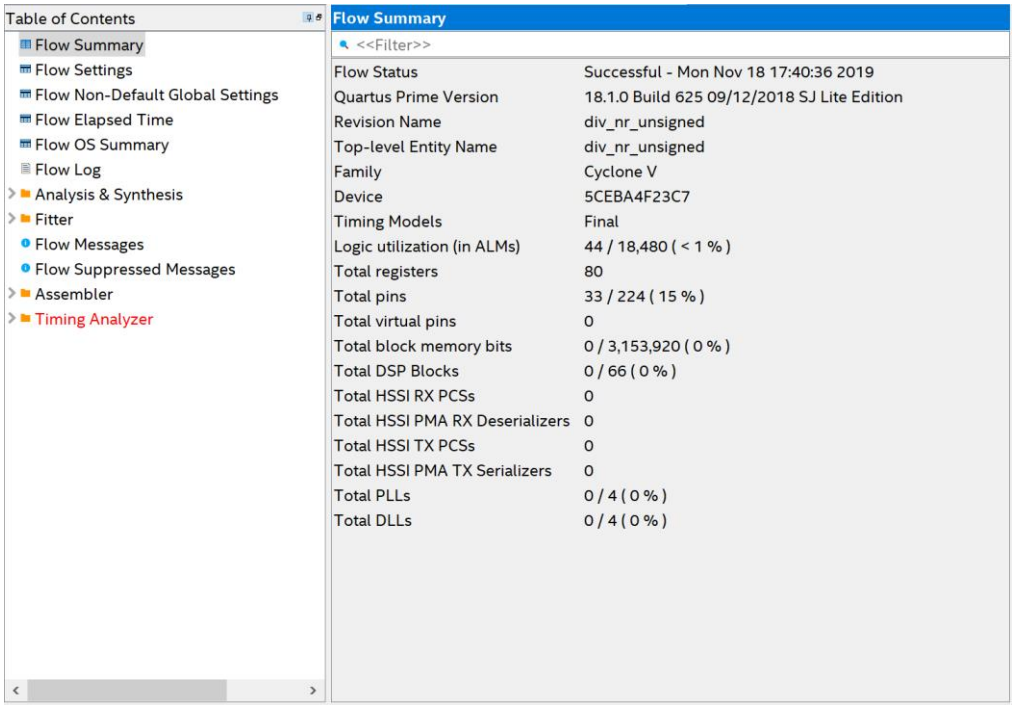


Figura 11: Utilización división sin restauración

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	44
2		
3	⌄ Combinational ALUT usage for logic	87
1	-- 7 input functions	0
2	-- 6 input functions	0
3	-- 5 input functions	0
4	-- 4 input functions	0
5	-- <=3 input functions	87
4		
5	Dedicated logic registers	80
6		
7	I/O pins	33
8		
9	Total DSP Blocks	0
10		
11	Maximum fan-out node	clk~input
12	Maximum fan-out	80
13	Total fan-out	517
14	Average fan-out	2.22

Figura 12: Parámetros división sin restauración

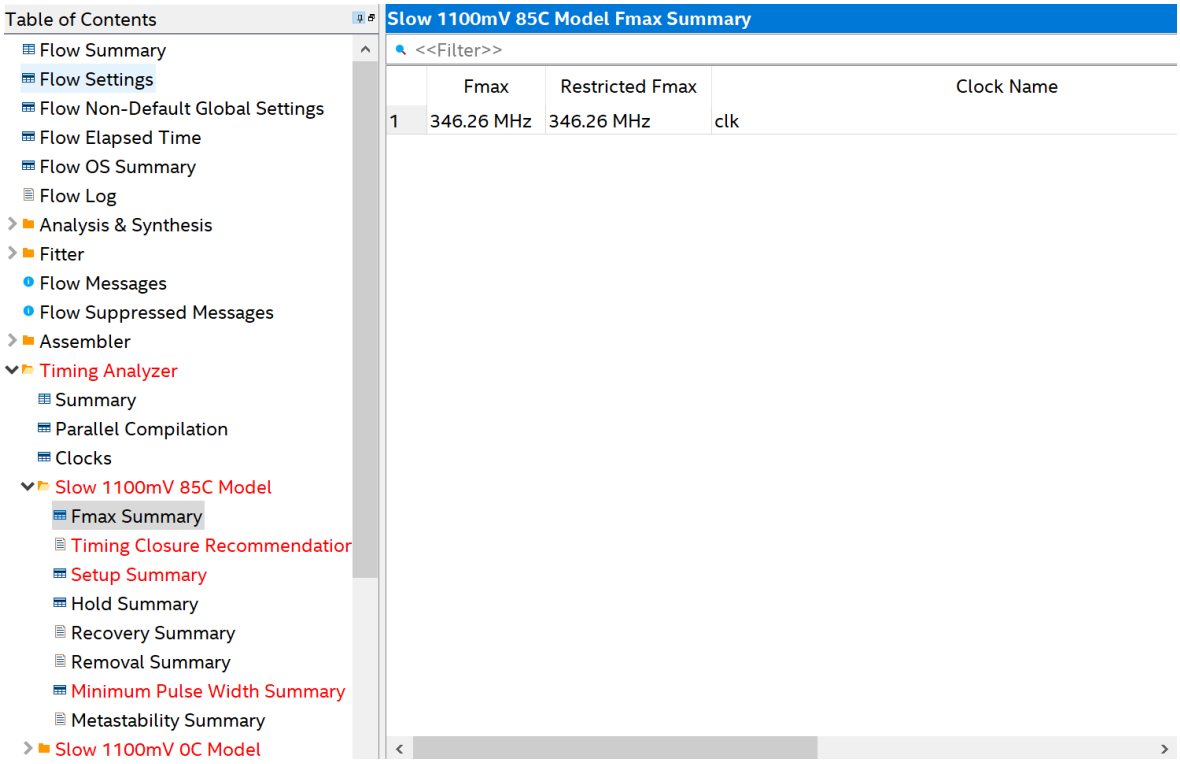


Figura 13: Frecuencia máxima división sin restauración

3.2.3 División rápida SRT

La figura 14 hace referencia al diseño al cual llegaríamos a partir del código desarrollado para este último algoritmo. Como se puede observar en la sección 'Anexos', consta de dos partes; el bloque principal div_SRT_r2 en el que se realiza la declaración de variables, se obtiene el resto y se realiza la conversión final del cociente y el bloque srt_step_2 en el que se obtiene el resultado de cada operación parcial.

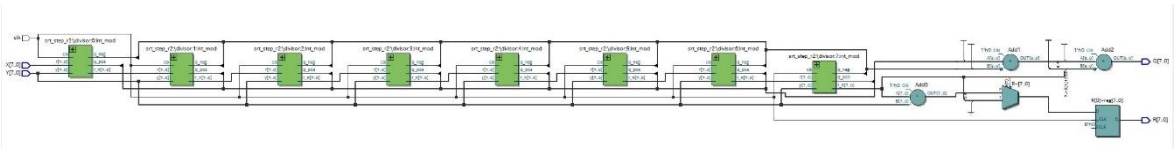


Figura 14: Diseño algoritmo SRT herramienta Quartus

Nuevamente, en las figuras 15, 16 y 17 tenemos la utilización y los recursos del algoritmo de división SRT.

Flow Status	Successful - Wed Jan 29 17:55:53 2020
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	div_SRT_r2
Top-level Entity Name	div_SRT_r2
Family	Cyclone V
Device	5CEBA4F23C8
Timing Models	Final
Logic utilization (in ALMs)	41 / 18,480 (< 1 %)
Total registers	72
Total pins	33 / 224 (15 %)
Total virtual pins	0
Total block memory bits	0 / 3,153,920 (0 %)
Total DSP Blocks	0 / 66 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 4 (0 %)

Figura 15: Utilización división SRT

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	41
2		
3	⌄ Combinational ALUT usage for logic	82
1	-- 7 input functions	0
2	-- 6 input functions	0
3	-- 5 input functions	0
4	-- 4 input functions	48
5	-- <=3 input functions	34
4		
5	Dedicated logic registers	72
6		
7	I/O pins	33
8		
9	Total DSP Blocks	0
10		
11	Maximum fan-out node	clk~input
12	Maximum fan-out	72
13	Total fan-out	553
14	Average fan-out	2.51

Figura 16: Parámetros división SRT

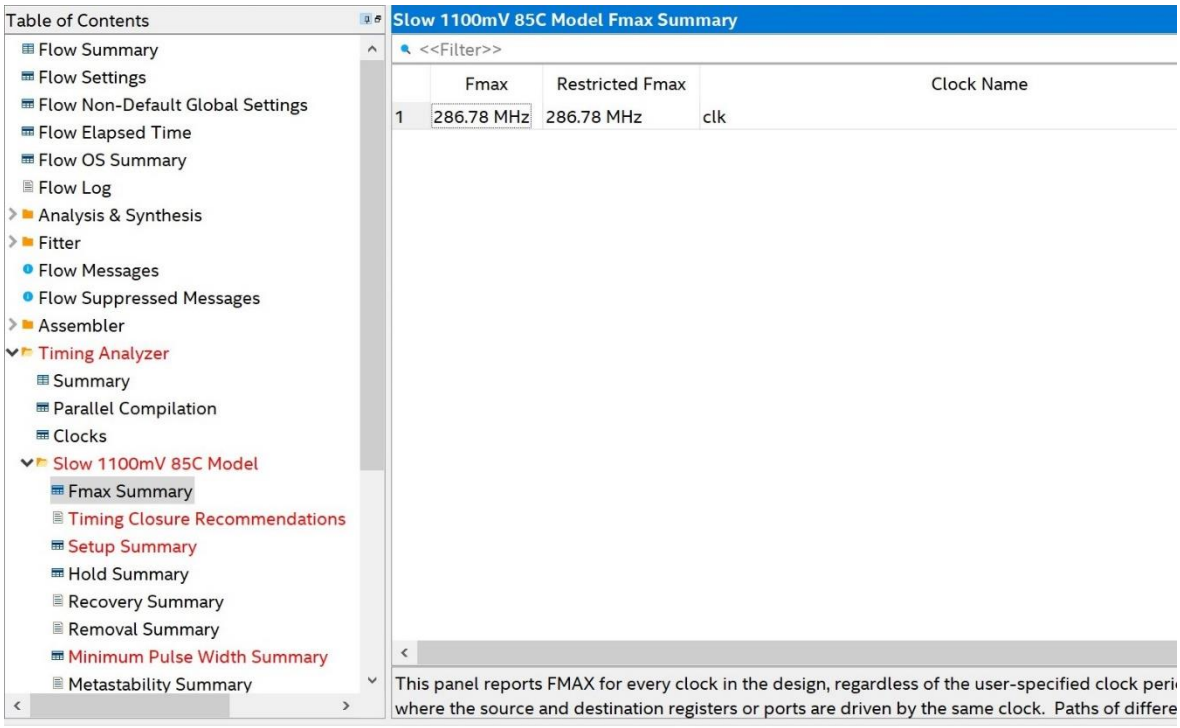


Figura 17: Frecuencia máxima división SRT

4 Análisis de resultados

Es importante mencionar que para todas las simulaciones que se van a ver de ahora en adelante se ha utilizado la herramienta de Quartus ***'University Program VWF (Vector Waveform File)'.***

4.1 División con restauración

Las figuras 18 y 19 muestran, de entre todas las operaciones de prueba realizadas, algunos ejemplos de cómo se comportaría este algoritmo.

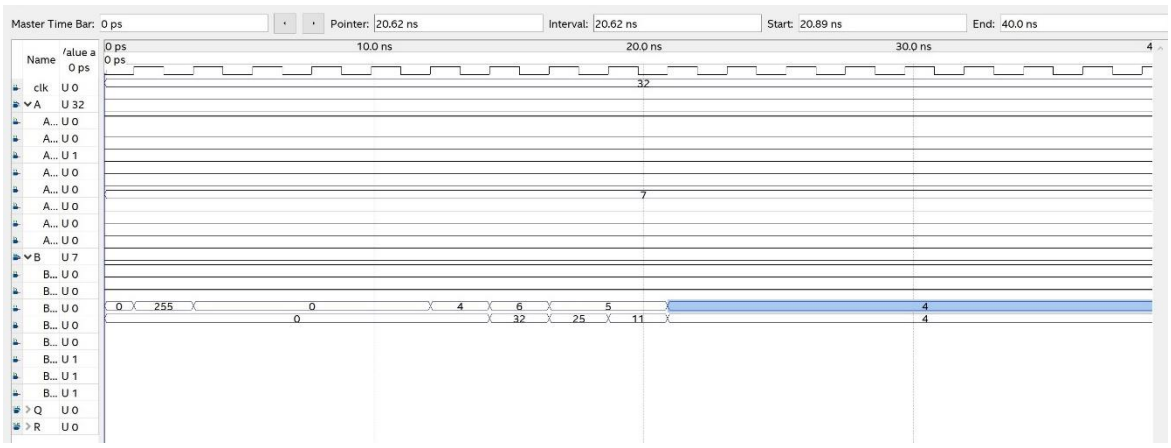


Figura 18: Ejemplo 1 división con restauración

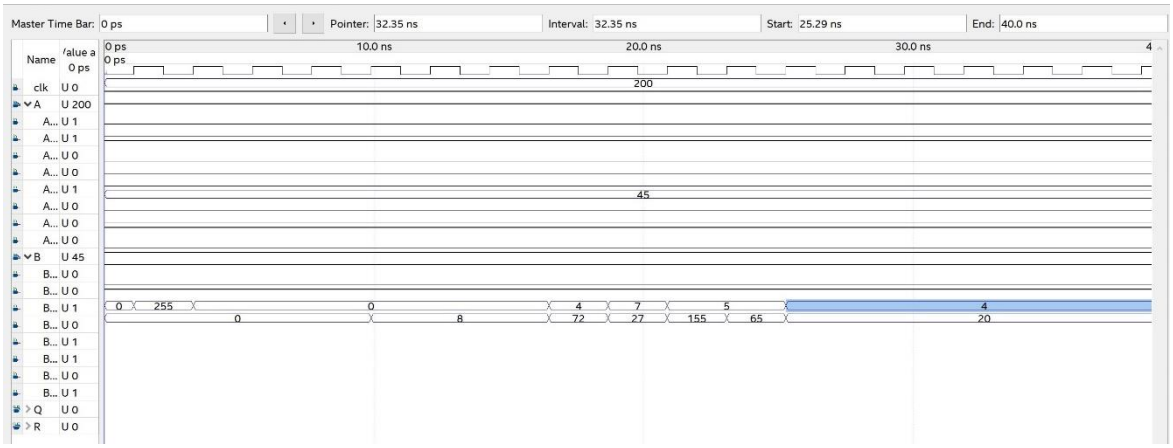


Figura 19: Ejemplo 2 división con restauración

Puesto que para el algoritmo de división con restauración tenemos varios aspectos a analizar más allá de los resultados obtenidos para el cociente y el resto de las operaciones y con el objetivo de simplificar la visualización de los datos, la tabla 4 recoge el tiempo y el número de ciclos que tarda el algoritmo en realizar cada una de las operaciones escogidas para el análisis.

Dividendo	Divisor	Cociente			Resto		
		Valor	Tiempo (ns)	Ciclos	Valor	Tiempo (ns)	Ciclos
1	1	1	3,3	2	0	-	-
2	1	2	3,3	2	0	-	-
2	2	1	7,7	4	0	-	
10	3	3	12,1	6	1	14,29	7
10	5	2	14,29	7	0	-	-
16	4	4	18,69	9	0	-	-
32	7	4	20,89	10	4	-	-
128	2	64	34,09	16	0	-	-
200	4	50	31,89	15	0	-	-
220	3	73	29,69	14	1	34,09	16
127	6	21	23,09	11	1	27,49	13
255	32	7	16,49	8	31	25,29	12
144	16	9	23,09	11	0	27,49	13
128	6	21	25,29	12	2	29,69	14
200	45	4	25,29	12	20	25,29	12
186	2	93	29,69	14	0	34,09	16

Tabla 4: Resultados división con restauración

4.2 División sin restauración

Siguiendo el análisis del algoritmo anterior, para el algoritmo de división sin restauración hemos empleado las figuras 20 y 21 para representar las simulaciones.

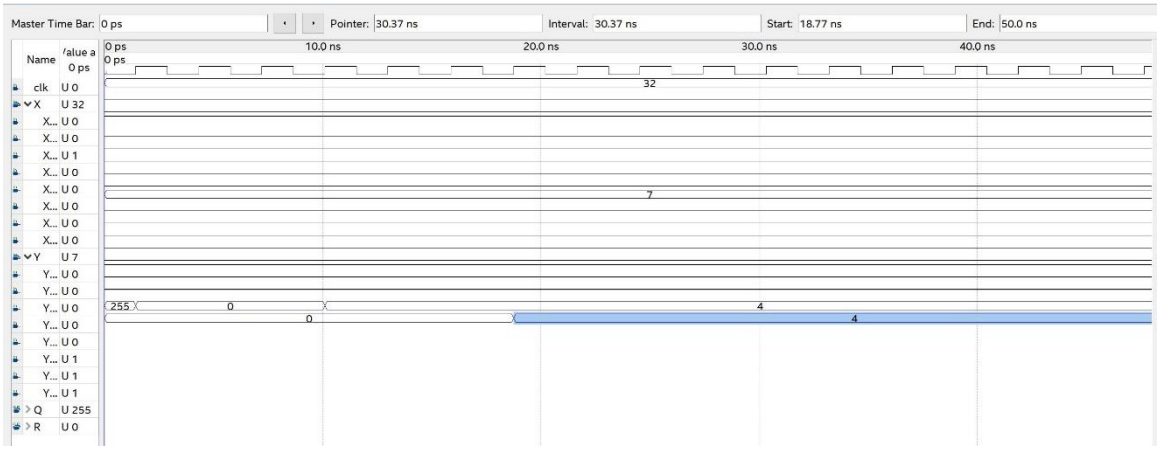


Figura 20: Ejemplo 1 división sin restauración

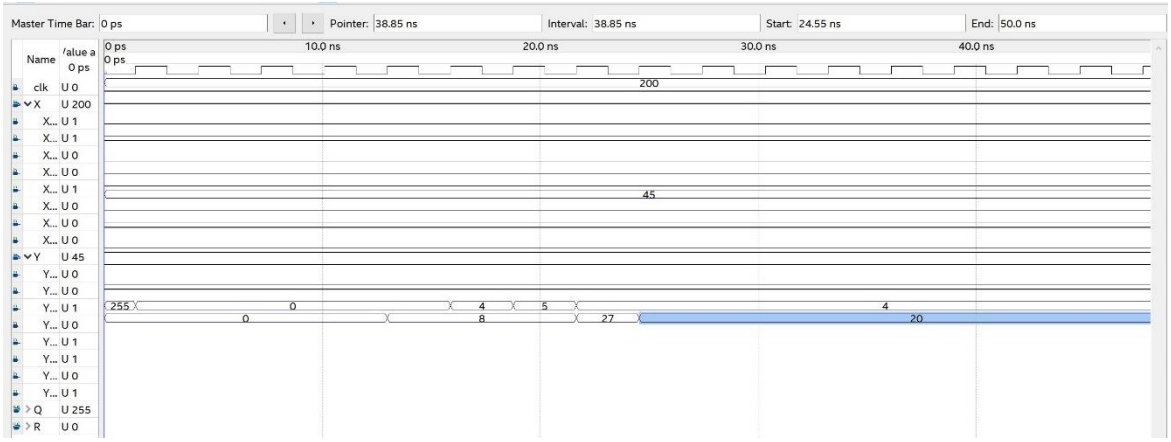
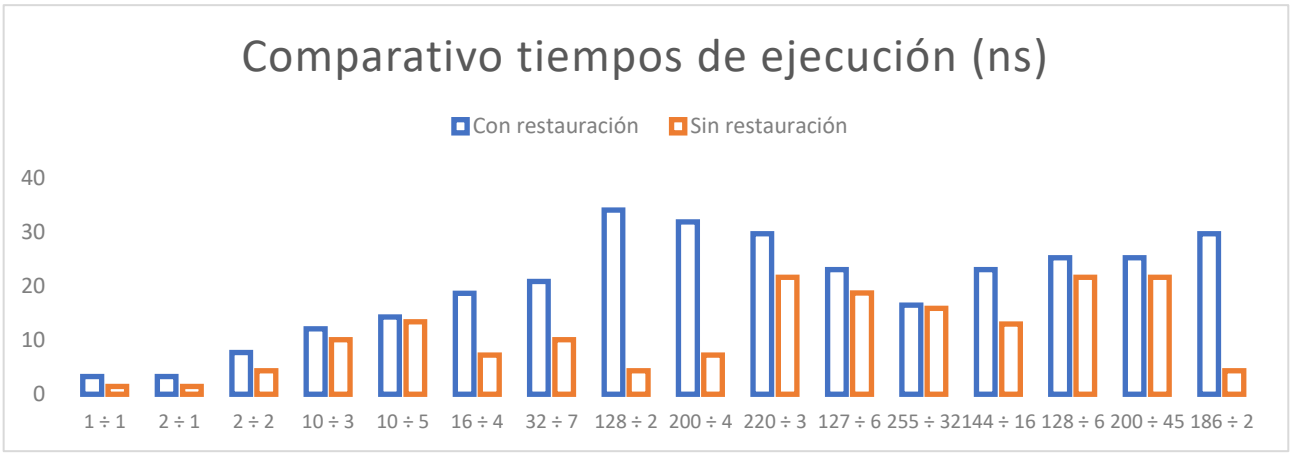


Figura 21: Ejemplo 2 división sin restauración

Al igual que para el algoritmo de división con restauración, para este algoritmo hemos representado los datos mediante la tabla 5.

Dividendo	Divisor	Cociente			Resto		
		Valor	Tiempo (ns)	Ciclos	Valor	Tiempo (ns)	Ciclos
1	1	1	1,44	< 1	0	-	-
2	1	2	1,44	< 1	0	-	-
2	2	1	4,33	2	0	-	
10	3	3	10,11	4	1	13	5
10	5	2	13,42	5	0	13	5
16	4	4	7,22	3	0	-	-
32	7	4	10,11	4	4	18,77	7
128	2	64	4,33	2	0	-	-
200	4	50	7,22	3	0	-	-
220	3	73	21,66	8	1	24,55	9
127	6	21	18,77	7	1	21,66	8
255	32	7	15,88	6	31	-	-
144	16	9	13,0	5	0	-	-
128	6	21	21,66	8	2	24,55	9
200	45	4	21,66	8	20	24,55	9
186	2	93	4,33	2	0	-	-

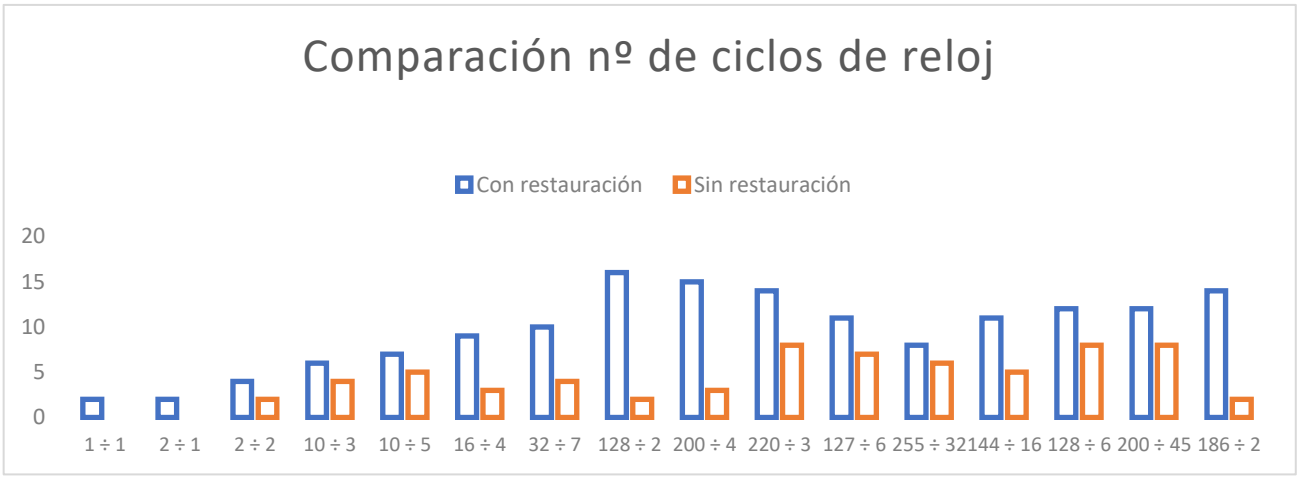
Tabla 5: Resultados división sin restauración



Gráfica 1: Comparación tiempos

A la vista de los resultados en la gráfica 1, para todas las operaciones analizadas, si nos ceñimos solo en los tiempos de ejecución, podemos concluir que el algoritmo de división sin restauración es más eficiente.

Nota 2: Para el análisis de la gráfica 1, hay que tener en cuenta que la frecuencia de reloj de los dos algoritmos es distinta.



Gráfica 2: Comparación ciclos

A la vista de la gráfica 2, podemos concluir también que el algoritmo sin restauración es más eficiente en cuanto al número de ciclos empleados en realizar cada operación.

4.3 División SRT

Puesto que, para este método, no se pueden elegir arbitrariamente los valores a dividir, y con el objetivo de comparar los resultados con los obtenidos en los apartados anteriores, vamos a implementar solo aquellas divisiones vistas anteriormente que se adapten también a este algoritmo. Esto queda reflejado en las Figueras 22 y 23.

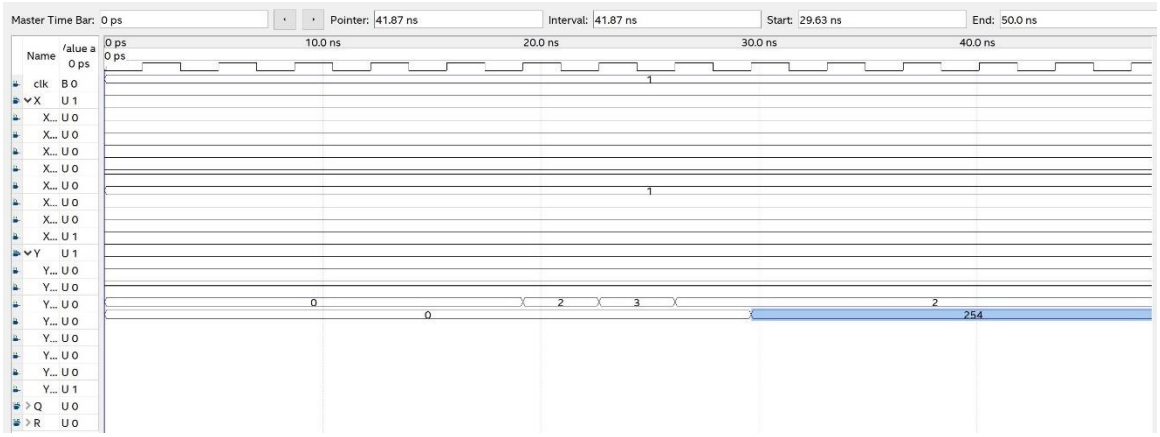


Figura 22: Ejemplo 1 división SRT

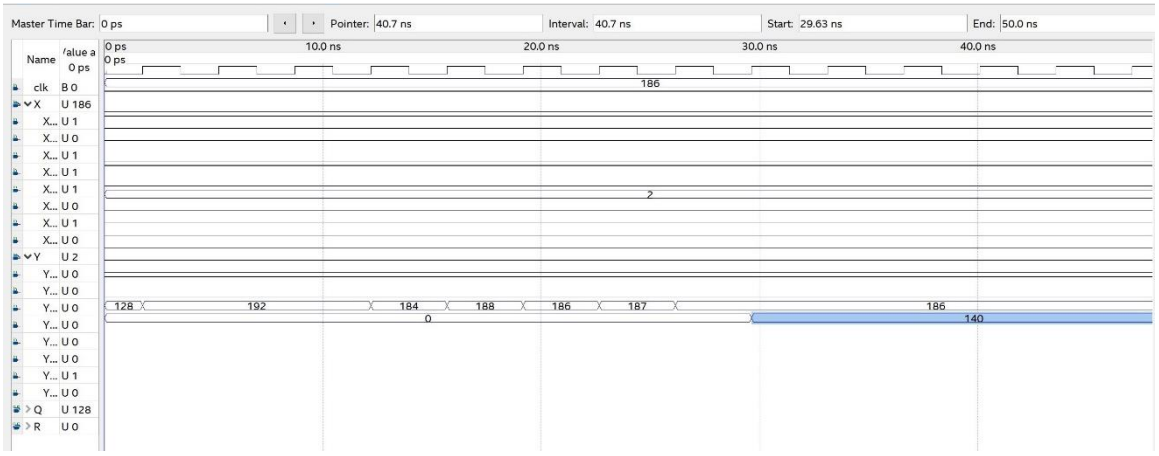


Figura 23: Ejemplo 2 división SRT

A la espera de realizar un análisis global de los resultados obtenidos en los diferentes métodos, estás dos figuras del algoritmo SRT nos dan una pista para saber que, al menos en términos de latencia, seguramente este algoritmo no sea el más adecuado. Esto se debe a que, como ya hemos comentado antes, los valores no se pueden elegir arbitrariamente ya

que es necesario que el dividendo sea menor o igual que el divisor para obtener un resultado correcto. Siempre y cuando se cumpla esta condición, este algoritmo será el más completo de todos los analizados.

4.4 Comparación de resultados

En esta sección vamos a pasar a comparar todos los resultados obtenidos de forma individual en las secciones anteriores con el objetivo de intentar definir cuál es el algoritmo óptimo de todos los vistos.

En la tabla 6, aparecen los distintos algoritmos con las frecuencias de reloj a las que trabajan.

	División síncrona	División con restauración	División sin restauración	División SRT
Frecuencia Máxima (MHz)	580.05 Mhz	454,75 MHz	346,26 MHz	286,78 MHz
Frecuencia Máxima (ns)	1,723 ns	2,199 ns	2,888 ns	3,486 ns
Ciclos 1 ÷ 1	< 1 ciclo	2 ciclos	< 1 ciclo	8 ciclos
Ciclos 186 ÷ 2	< 1 ciclo	16 ciclos	2 ciclos	8 ciclos
(Área) Nº total de Pins	32 / 224 (15%)	33 / 224 (15%)	33 / 224 (15%)	33 / 224 (15%)
Nº total de Registros	32	144	80	72
Utilización en ALMs	92 / 18.480 (< 1%)	73 / 18.480 (< 1%)	44 / 18.480 (< 1%)	41 / 18.480 (<1%)

Tabla 6: Comparación de resultados

Si nos centrásemos únicamente en los datos analizados en la tabla 6, podríamos concluir que cada uno de los algoritmos destacaría en uno de los parámetros analizados. No obstante, si tenemos en cuenta el conjunto global de todos los campos a valorar, podríamos concluir que quizás el algoritmo de división sin restauración sea el más adecuado para nuestro objetivo inicial.

5 Conclusiones finales

5.1 Aspectos técnicos

En este trabajo se han estudiado 3 alternativas para la operación de división binaria. Las gráficas 1 y 2 resumen los principales resultados entre los dos métodos más importantes.

A raíz de esos resultados podemos concluir que el algoritmo que mejor se adaptaría al cometido de este TFG, en términos globales, sería el algoritmo de división sin restauración.

Desde el punto de vista de velocidad, y teniendo en cuenta solamente las frecuencias máximas, tendríamos al algoritmo con restauración como el más rápido.

Para el resto de los casos, el algoritmo más rápido sería el algoritmo sin restauración pues disminuye considerablemente el número de ciclos empleados en cada división.

Por último, como hemos comentado anteriormente, para aquellas operaciones en las que d > D, el algoritmo más apropiado es el algoritmo SRT.

En cuanto a ocupación de recursos el algoritmo que mejor se ajusta es el algoritmo SRT.

5.2 Aspectos educativos

Desde el punto de vista de formación, este TFG me ha permitido:

Conocer y practicar la tecnología de Intel-Altera (En la EPS UAM sólo se utiliza Xilinx), ampliar mis conocimientos de aritmética y estudiar el tema con algo más de profundidad en diversos libros avanzados y, por último, ampliar mis conocimientos de VHDL.

5.3 Trabajo Futuro

Debido al cierre de actividades (tanto en la UAM como en UDELAR) no fue posible incorporar la variable consumo de energía. En el futuro se espera ampliar el estudio para abarcar esta variable.

Referencias

[1] J. M. Menegáz and D. S. L. Simonetti, "A review of the main inverter topologies applied on the integration of renewable energy resources to the grid," XI Brazilian Power Electronics Conference, Praiamar, 2011, pp. 963-969. doi: 10.1109/COBEP.2011.6085334

[2] GOSLING. (2013). Design of Arithmetic Units for Digital Computers. New York, NY: Springer.

[3] Deschamps, J., Bioul, G., & Sutter, G. (2006). *Synthesis of arithmetic circuits*. Hoboken, NJ: Wiley-Interscience.

[4] Ercegovac, M., & Lang, T. (2008). *Digital arithmetic*. San Francisco, CA: Morgan Kaufmann.

[5] Kulisch, U. (2002). Advanced arithmetic for the digital computer. Wien: Springer.

[6] Oberman, R. (1979). *Digital circuits for binary arithmetic*. London: Macmillan.

[7] Parhami, B. (2010). *Computer arithmetic*. New York: Oxford University Press.

[8] Stine, J. Digital Computer Arithmetic Datapath Design Using Verilog HDL.

[9] (2020). Retrieved 1 February 2020, from http://digital.csic.es/bitstream/10261/86812/1/CMOS%20nanom%C3%A9tricas.pdf

[10] Basic Binary Division: The Algorithm and the VHDL Code - Technical Articles. (2020). Retrieved 1 February 2020, from https://www.allaboutcircuits.com/technical-articles/basic-binary-division-the-algorithm-and-the-vhdl-code/

[11] divider, V., & Zilmer, M. (2020). VHDL 4-bit binary divider. Retrieved 1 February 2020, from https://stackoverflow.com/questions/20203354/vhdl-4-bit-binary-divider

[12] Technologies, T. (2020). Terasic - All FPGA Main Boards - Cyclone V - DE0-CV Board. Retrieved 1 February 2020, from https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=921

[13] (2020). Retrieved 10 May 2020, from http://www.esi.uclm.es/www/isanchez/eco0910/alu.pdf

Glosario

TFG	Trabajo Fin de Grado
EPS	Escuela Politécnica Superior
EEG	Encefalografía
SRT	*Shortest Remaining Time*
Mbps	*Megabits per second*
VHDL	*very-high-speed integrated circuits hardware description language*
I/O	*Input/Output*
RAM	*Randome Access Memory*
TJ	*Junction Temperature*
BGA	*Ball Grid Array*
FBGA	*Fine-Pitch Ball Grid Array*
DSP	*Digital Signal Processing*
ALU	*Arithmetic Logic Unit*
FPGA	*Field Programmable Gate Array*
IEEE	*Institute of Electrical and Electronics Engineers*

Anexos

Códigos VHDL utilizados para el análisis práctico de algoritmos

Algoritmo con restauración

Bloque div_rest_nat:

```
package mypackage is
      constant NBITS :INTEGER := 8;
      constant MBITS :INTEGER := 8;
end mypackage;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.mypackage.all;

entity div_rest_nat is
    port (
        clk: in std_logic;
              A: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
        B: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
        Q: out STD_LOGIC_VECTOR (NBITS-1 downto 0);
              R: out STD_LOGIC_VECTOR (MBITS-1 downto 0)
     );
end div_rest_nat;

architecture div_arch of div_rest_nat is

component restoring_cell is
    port (
              clk: in std_logic;
        a: in STD_LOGIC_VECTOR (MBITS downto 0);
        b: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
              q: out STD_LOGIC;
              r: out STD_LOGIC_VECTOR (MBITS downto 0)
     );
end component;

type conections is array (0 to NBITS-1) of STD_LOGIC_VECTOR (MBITS downto
0);
Signal wires_in, wires_out: conections;
Signal zeros: STD_LOGIC_VECTOR (MBITS-1 downto 0);

begin

 zeros <= (others => '0');
 wires_in(0) <= zeros & A(NBITS-1);
```

```
 divisor: for i in 0 to NBITS-1 generate
    rest_cell: restoring_cell port map (a => wires_in(i),
                  clk => clk,
                   b => B,
                 q => Q(NBITS-I-1),
                    r => wires_out(i));

 end generate;

 wires_conections: for i in 0 to NBITS-2 generate
        wires_in(i+1) <= wires_out(i)(MBITS-1 downto 0) & A(NBITS-i-2);

 end generate;

  R <= wires_out(NBITS-1)(MBITS-1 downto 0);

end div_arch;
```

Bloque restoring_cell:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.mypackage.all;

entity restoring_cell is
    port (
               clk: in std_logic;
         a: in STD_LOGIC_VECTOR (MBITS downto 0);
         b: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
               q: out STD_LOGIC;
               r: out STD_LOGIC_VECTOR (MBITS downto 0)
     );
end restoring_cell;

architecture cel_arch of restoring_cell is

signal subst: STD_LOGIC_VECTOR (MBITS downto 0);

begin

multiplexer: process (a,b,subst,clk)
begin

if rising_edge(clk) then
         subst <= a - b;
         if subst(MBITS) = '1' then
              r <= a;
         else
              r <= subst;
         end if;
        q <= not subst(MBITS);
         end if;
```

```
end process;


end cel_arch;
```

Algoritmo sin restauración

Bloque div_nr_unsigned:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.mypackage.all;

entity div_nr_unsigned is
    port (
              clk: in std_logic;
        X: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
        Y: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
        Q: out STD_LOGIC_VECTOR (NBITS-1 downto 0);
              R: out STD_LOGIC_VECTOR (MBITS-1 downto 0)
     );
end div_nr_unsigned;

architecture mult_arch of div_nr_unsigned is

COMPONENT cond_adder
      PORT(
            clk: in std_logic;
            a : IN std_logic_vector(MBITS-1 downto 0);
            b : IN std_logic_vector(MBITS-1 downto 0);
            x_i : IN std_logic;
            r : OUT std_logic_vector(MBITS-1 downto 0)
            );
END COMPONENT;

component nr_cell is
    port (
              clk: in std_logic;
        a: in STD_LOGIC_VECTOR (MBITS downto 0);
        b: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
        q: in STD_LOGIC;
              r: out STD_LOGIC_VECTOR (MBITS downto 0)
     );
end component;

type conections is array (0 to NBITS-1) of STD_LOGIC_VECTOR (MBITS downto
0);
Signal wires_in, wires_out: conections;

Signal  QQ: STD_LOGIC_VECTOR (NBITS downto 0);
```

```
signal  adjust: STD_LOGIC;
Signal  zeros: STD_LOGIC_VECTOR (MBITS-1 downto 0);

begin


 QQ(NBITS) <= '0';
 zeros <= (others => '0');
 wires_in(0) <= zeros & X(NBITS-1);

 divisor: for I in 0 to NBITS-1 generate
    int_mod: nr_cell port map (a => wires_in(I),
                clk => clk,
                   b => Y,
                   q => QQ(NBITS-I),
                   r => wires_out(i) );
  end generate;


 wires_conections: for I in 0 to NBITS-2 generate
    QQ(NBITS-i-1) <= wires_out(i)(MBITS);
    wires_in(i+1) <= wires_out(i)(MBITS-1 downto 0) & X(NBITS-I-2);
  end generate;

 adjust <= (wires_out(NBITS-1)(MBITS));


 final_adjust: cond_adder port map (
                  clk => clk,
                   a => wires_out(NBITS-1)(MBITS-1 downto 0),
                   b => Y,
                   x_i => adjust,
                   r => R);

 Q(NBITS-1 downto 1) <= not QQ(NBITS-1 downto 1);
 Q(0) <= not adjust;


end mult_arch;
```

Bloque mypack:

```
package mypackage is
      constant NBITS :INTEGER := 8;
      constant MBITS :INTEGER := 8;
end mypackage;
```

Bloque cond_adder:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.mypackage.all;

entity cond_adder is
    port (
              clk: in std_logic;
        a: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
        b: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
        x_i: in STD_LOGIC;
              r: out STD_LOGIC_VECTOR (MBITS-1 downto 0)
     );
end cond_adder;

architecture cond_adder_arch of cond_adder is

begin

conditional_adder: process (x_i,a,b,clk)
begin

        if rising_edge(clk) then
        if x_i = '1' then
            r <= a + b;
        else
            r <= a;
        end if;
        end if;

end process;


end cond_adder_arch;
```

Bloque nr_cell:

Este bloque se encarga de, en base al resultado obtenido en el cociente, realizar las sumas o restas correspondientes.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.mypackage.all;

entity nr_cell is
    port (
                clk: in std_logic;
                a: in STD_LOGIC_VECTOR (MBITS downto 0);
                b: in STD_LOGIC_VECTOR (MBITS-1 downto 0);
                q: in STD_LOGIC;
                r: out STD_LOGIC_VECTOR (MBITS downto 0)
     );
end nr_cell;

architecture nr_cel_arch of nr_cell is

begin

adder_subtracter: process (clk,q,a,b)
begin

if rising_edge(clk) then
        if q = '1' then
                r <= a + b;
        else
                r <= a - b;
        end if;
        end if;
end process;


end nr_cel_arch;
```

Algoritmo SRT:

Bloque div_SRT_r2:

```
package mypackage is
      constant NBITS :INTEGER :=8;
      constant PBITS :INTEGER :=8;
end mypackage;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.mypackage.all;
library UNISIM;
use UNISIM.ALL;

entity div_SRT_r2 is
    port (
              clk: in std_logic;
         X: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
         Y: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
         Q: out STD_LOGIC_VECTOR (PBITS-1 downto 0);
             R: out STD_LOGIC_VECTOR (NBITS-1 downto 0)
     );
end div_SRT_r2;

architecture srt_arch of div_SRT_r2 is

component srt_step_r2 is
    port (
       clk: in std_logic;
        r: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
        y: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
        q_pos, q_neg: out STD_LOGIC;
              r_n: out STD_LOGIC_VECTOR (NBITS-1 downto 0)
     );
end component;

type conections is array (0 to PBITS) of STD_LOGIC_VECTOR (NBITS-1 downto
0);
Signal wires: conections;
signal adjust: STD_LOGIC;
signal Q_pos, Q_neg: STD_LOGIC_VECTOR (PBITS-1 downto 0);

begin

 wires(0) <= X;
 divisor: for I in 0 to PBITS-1 generate
    int_mod: srt_step_r2 port map (r => wires(i),
                   clk=>clk,
                   y => Y,
                   q_neg => Q_neg(PBITS-I-1),
                   q_pos => Q_pos(PBITS-I-1),
                   r_n => wires(i+1) );
  end generate;
```

```
  adjust <= (wires(PBITS)(NBITS-1));

  correction_step: process (adjust, wires(PBITS),clk)
  begin
  if rising_edge(clk) then
    if adjust = '0' then
         R <= wires(PBITS)(NBITS-1 downto 0);
    else
         R <= wires(PBITS)(NBITS-1 downto 0) + Y;
    end if;
       end if;
  end process;

  Q <= Q_pos - Q_neg - adjust;

end srt_arch;
```

Bloque srt_step_r2:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.mypackage.all;
library UNISIM;
use UNISIM.ALL;

entity srt_step_r2 is
    port (
              clk: in std_logic;
        r: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
        y: in STD_LOGIC_VECTOR (NBITS-1 downto 0);
        q_pos, q_neg: out STD_LOGIC;
              r_n: out STD_LOGIC_VECTOR (NBITS-1 downto 0)
     );
end srt_step_r2;

architecture behavioural of srt_step_r2 is
signal r_x_2 : STD_LOGIC_VECTOR (NBITS-1 downto 0);
begin

r_x_2 <= r(NBITS-2 downto 0) & '0';

adder_subtracter: process (r,y,r_x_2,clk)
begin
if rising_edge(clk) then
      case r(NBITS-1 downto NBITS-2) is
      when "00" => r_n <= r_x_2;       q_pos <= '0'; q_neg <= '0';
            when "01" => r_n <= r_x_2 - y;      q_pos <= '1'; q_neg <=
'0';
            when "10" => r_n <= r_x_2 + y;      q_pos <= '0'; q_neg <=
'1';
            when "11" => r_n <= r_x_2;       q_pos <= '0'; q_neg <= '0';
            when others => NULL;
   end case;
      end if;
end process;
```

```
end behavioural;

architecture srt_cel_arch of srt_step_r2 is
signal r_x_2, y_and : STD_LOGIC_VECTOR (NBITS-1 downto 0);
signal q_n, q_p, r_xor : STD_LOGIC;

begin

r_xor <= r(NBITS-1) xor r(NBITS-2);
q_n <= r_xor and r(NBITS-1);
q_p <= r_xor and r(NBITS-2);
r_x_2 <= r(NBITS-2 downto 0) & '0';
ands_Y: for i in 0 to NBITS-1 generate
  Y_and(i) <= y(i) and r_xor;
end generate;

adder_subtracter: process (q_p,Y_and,r_x_2,clk)
begin
if rising_edge(clk) then
  if q_p = '0' then
    r_n <= r_x_2 + Y_and ;
  else
    r_n <= r_x_2 - Y_and ;
  end if;
  end if;
end process;
q_pos <= q_p;
q_neg <= q_n;
end srt_cel_arch;
```